RU Staying

Software Engineering
01:332:452
Report 2

By Group #11
Keya Patel
Zain Sayed
Mohammed Sapin
Purna Haque
Nga Man (Mandy) Cheng
Rameen Masood
Shilp Shah
Mathew Varghese
Thomas Tran
Eric Zhang

Github: https://github.com/mohammedsapin/RUStaying
Submission Date: March 17, 2019

Individual Contributions

	Interaction Diagrams	Class Diagram	System Architecture	Algorithms and Data Structures	User Interface Design and Implementation	Design of Tests	Project Management and Plan of Work
Keya	10%	10%	10%	10%	10%	10%	10%
Zain	10%	10%	10%	10%	10%	10%	10%
Mohammed	10%	10%	10%	10%	10%	10%	10%
Purna	10%	10%	10%	10%	10%	10%	10%
Nga Man	10%	10%	10%	10%	10%	10%	10%
Rameen	10%	10%	10%	10%	10%	10%	10%
Shilp	10%	10%	10%	10%	10%	10%	10%
Mathew	10%	10%	10%	10%	10%	10%	10%
Eric	10%	10%	10%	10%	10%	10%	10%
Thomas	10%	10%	10%	10%	10%	10%	10%

Table of Contents

Summary of Changes from Report 1

We switched persistent data storage from Sqlite to Firebase. We realized SQlite was serverless and it would be much harder to communicate data between different clients as SQlite only stores data locally. We would have to use a language such as PHP to implement a server in order for the database to communicate with different clients. Firebase has a much easier implementation because it stores data in the cloud and updates in realtime. Firebase provides a lot of public open source resources and APK making it very easy to use. Android Studio provides a lot of support as Firebase is easy to integrate and provides documentation on how to read and write data. Therefore we found it more convenient to use firebase instead of SQlite.

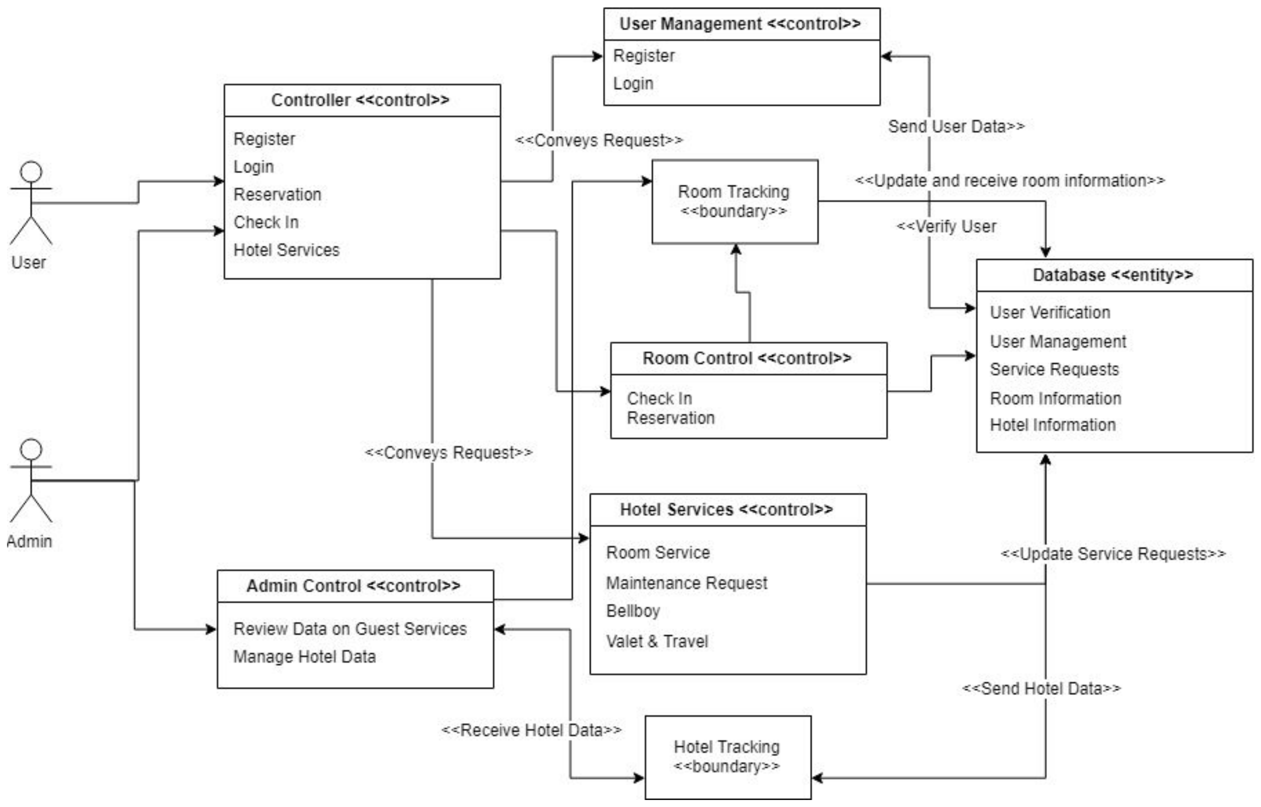
Chapter 1

Interaction Diagrams

1.1 Concept Definition and Responsibilities

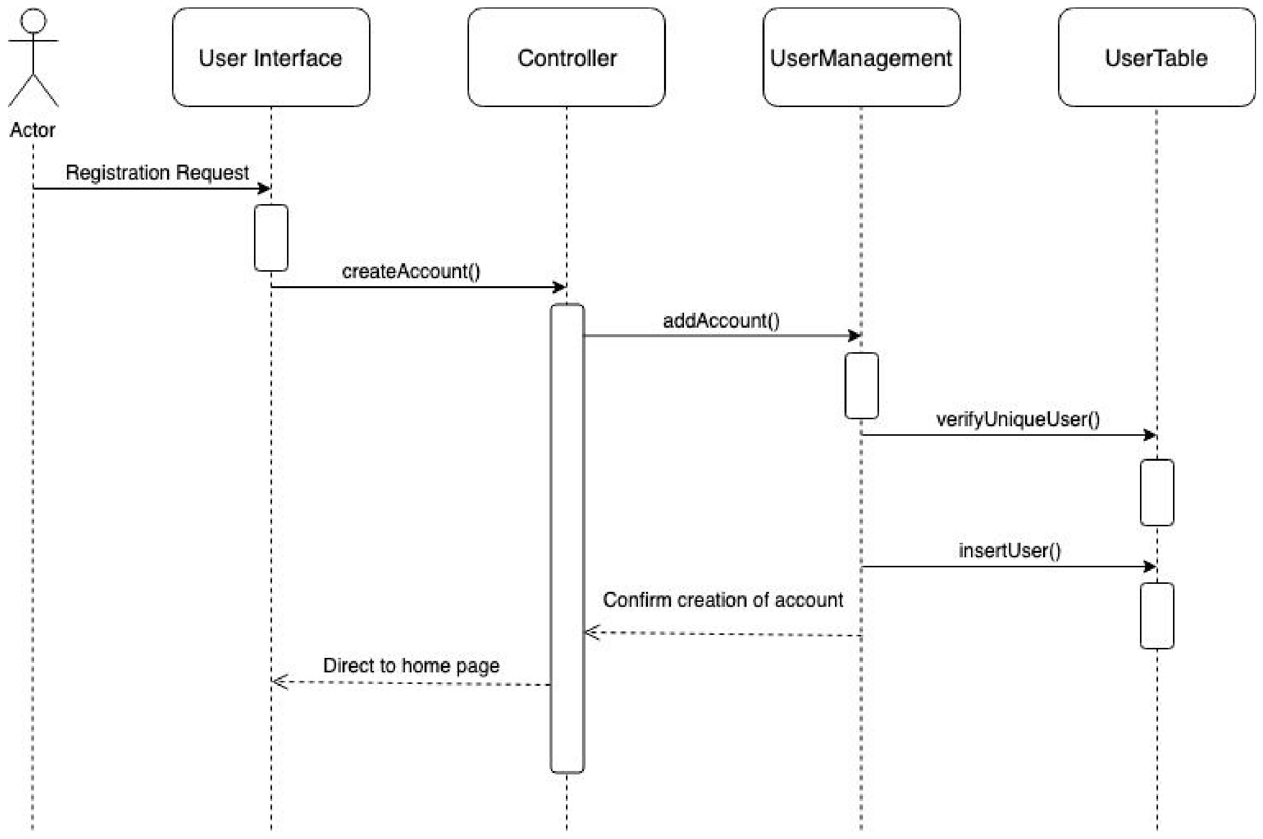
Responsibility Description	**Type**	**Concept Name**
RS1. Main source for all other subsystems to interact with. Coordinate actions and delegate work based on user interactions.	D	Controller
RS2. Verify if user credentials are valid	D	UserVerification
RS3. Create a new guest account	K	UserManagement
RS5. Store user account information (username, password, email, previous reservations etc…)	D	UserManagement
RS4. To allow user to make a reservation	D	RoomControl
RS6. To allow the guest to check-in for their hotel reservation	K	RoomControl
RS7. Guest can request room service	D	HotelServices
RS8. Guest can make maintenance requests for their room during their stay	D	HotelServices
RS9. Guest can request a bellboy for luggage	K	HotelServices
RS10. Guest can use the hotel car service	K	HotelServices
RS11. System will keep track of available / unavailable rooms	D	RoomTracking
RS12. System will keep track of each hotel service as it is requested	D	HotelTracking
RS13. Review all data on services used by guests	D	AdminControl
RS14. Present data in viewable manner to analysis and predictions	D	AdminControl

1.2 Domain Model Diagram



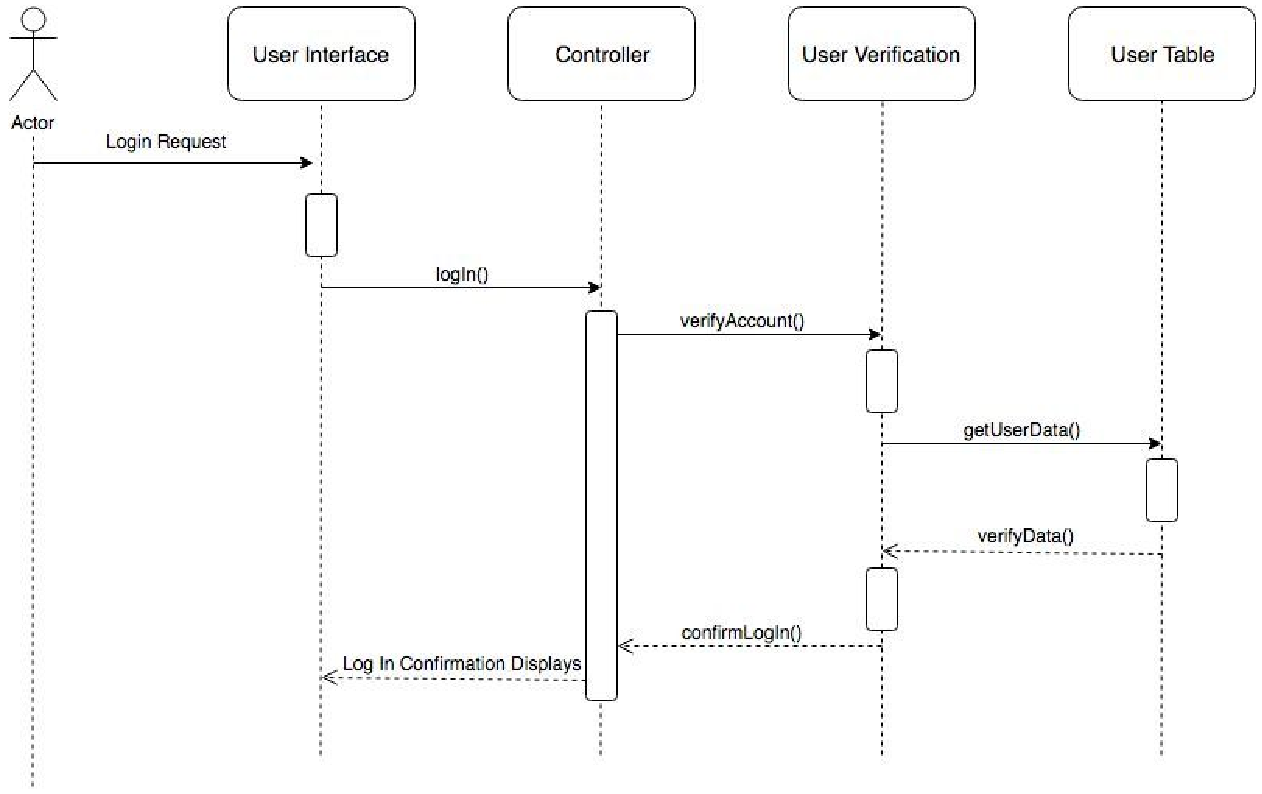
1.3 System Sequence Diagrams

Use Case 1 - Registration



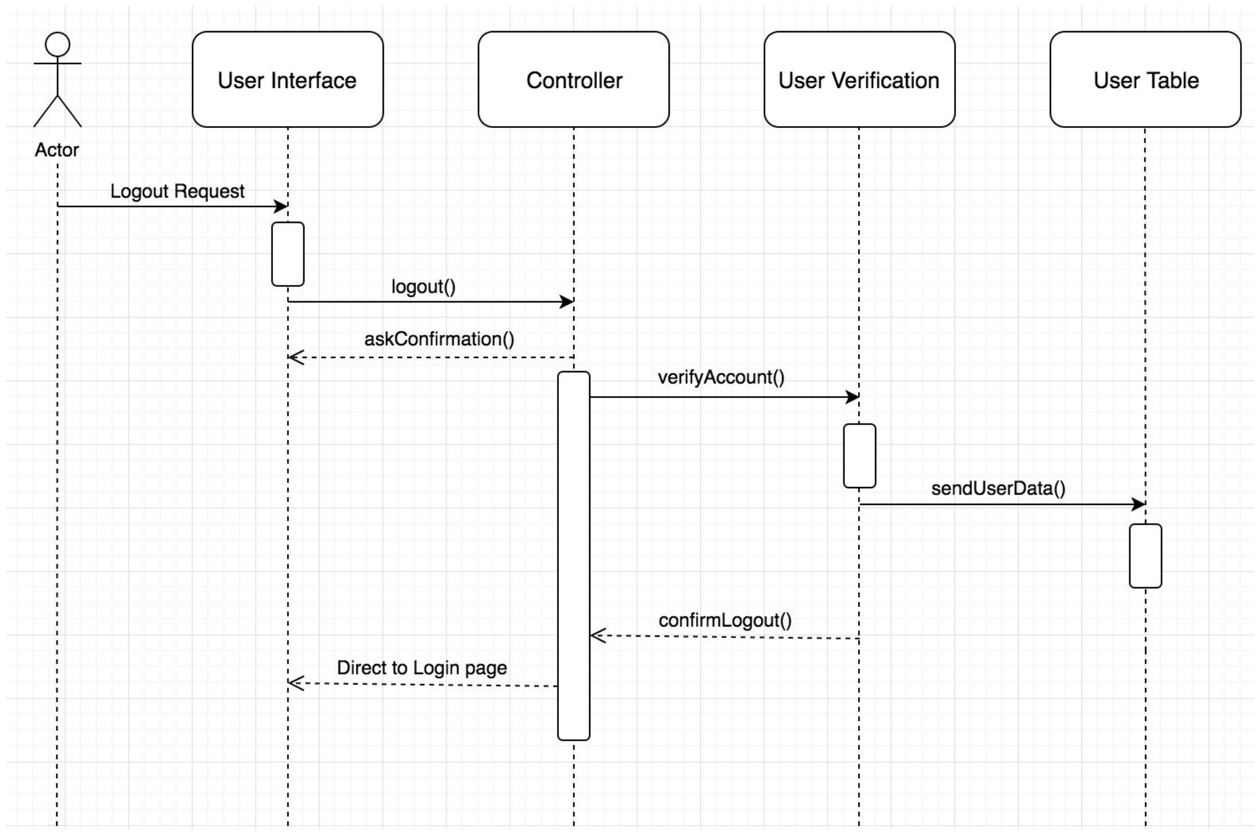
In this use case, the user can register for a new account. This is a relatively straight forward process. Once the user fills out the required information for a new account, the main controller will send the information to the UserManagement controller. The UserManagement controller is responsible for keeping track of all current and new accounts with the app RUStaying. It directly interacts with the database, specifically the UserTable, which is a table with all current accounts and their corresponding information. The UserTable concept was not listed in the domain model because we grouped all the specific database related entities. However, in these interaction diagrams, it is important to separate the different tables and information the database will be storing. The UserManagement controller will verify with the UserTable if this is a unique account and then process to insert the new user to the table. Once the main controller gets a confirmation that the user was created, it redirects the User Interface to the main home page so the user can access the hotel services through the app.

Use Case 2 - Login



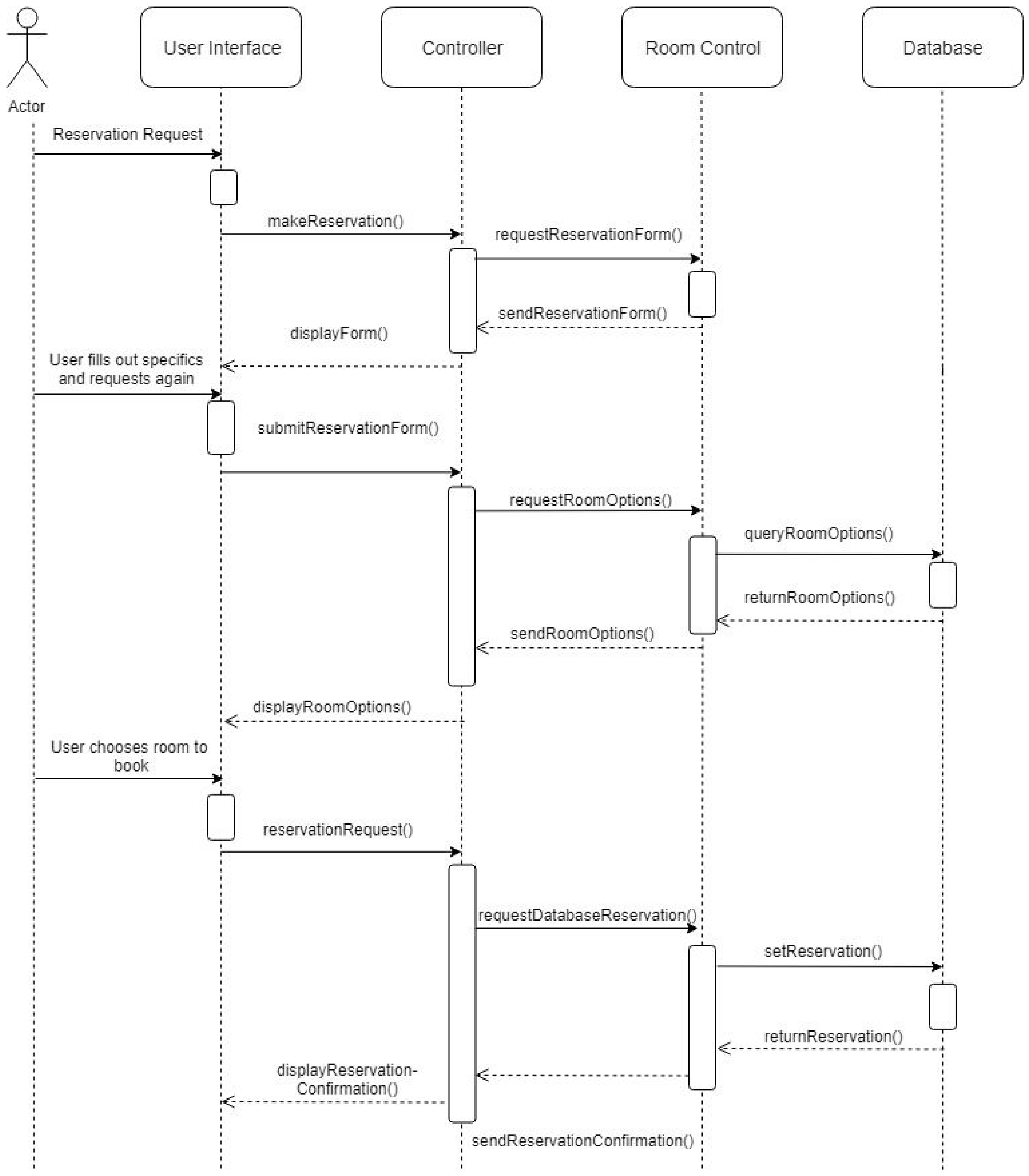
In this use case, the user can log into their account. Once the user inputs their username and password into the interface, the main controller will send the information into the UserVerification control. This is responsible for sending the data submitted by the user to the UserTable and verify if the information entered corresponds with an account that is in the database. If the account information is confirmed, the UserTable will send a verification to the UserVerification if the account information matches that of an existing account. The UserVerification will then confirm the login and allow the user to log in. If the information entered by the user does not correspond, then the user will be asked to input the login information again.

Use Case 3 - Logout



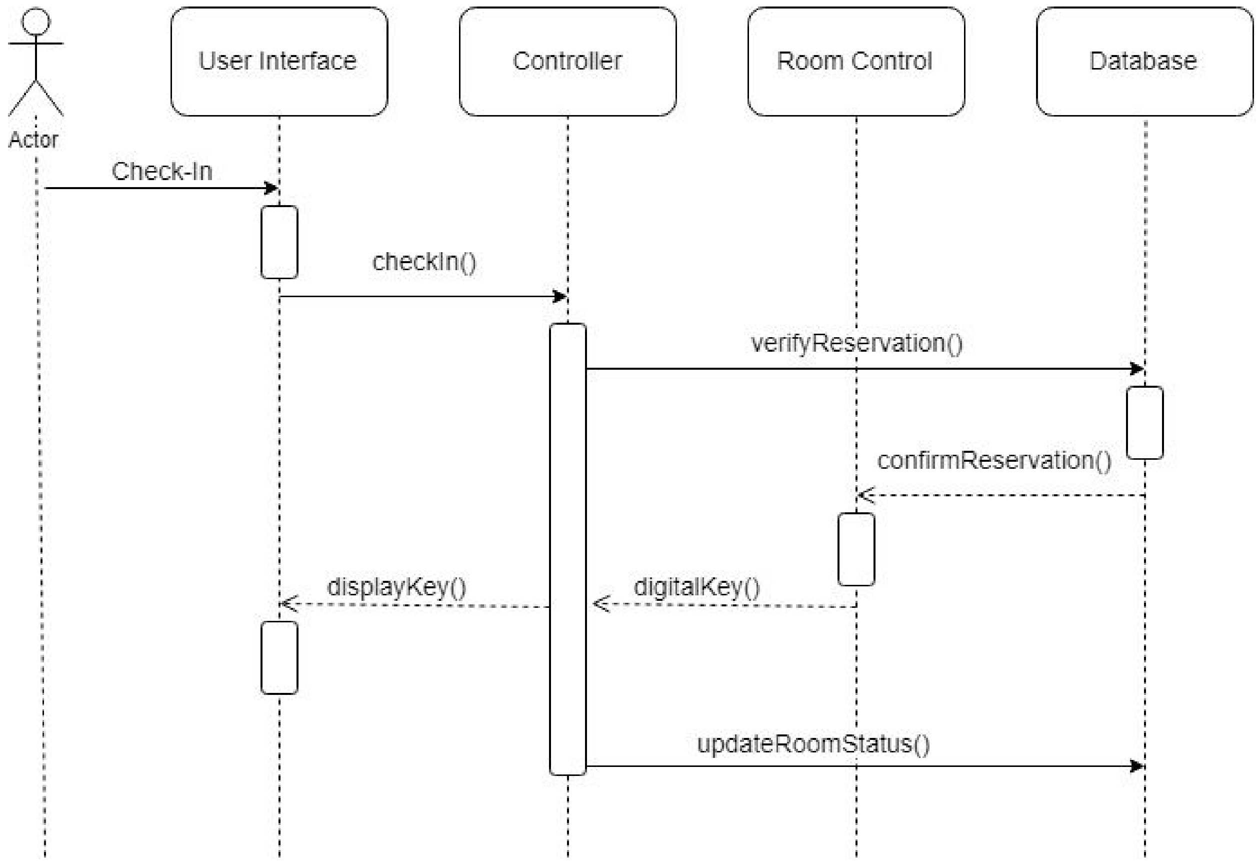
In this user case, the user is able to log out of an existing account. Once the user creates a logout request, the main controller will send a confirmation message back to the user to confirm logout. Then, the controller will send the information UserVerification control. User Verification will send the data submitted by the user to the User Table. The UserVerification will then confirm the logout and allow the user to logout of their account. The user will be redirected to the login page.

Use Case 4 - Make a Reservation



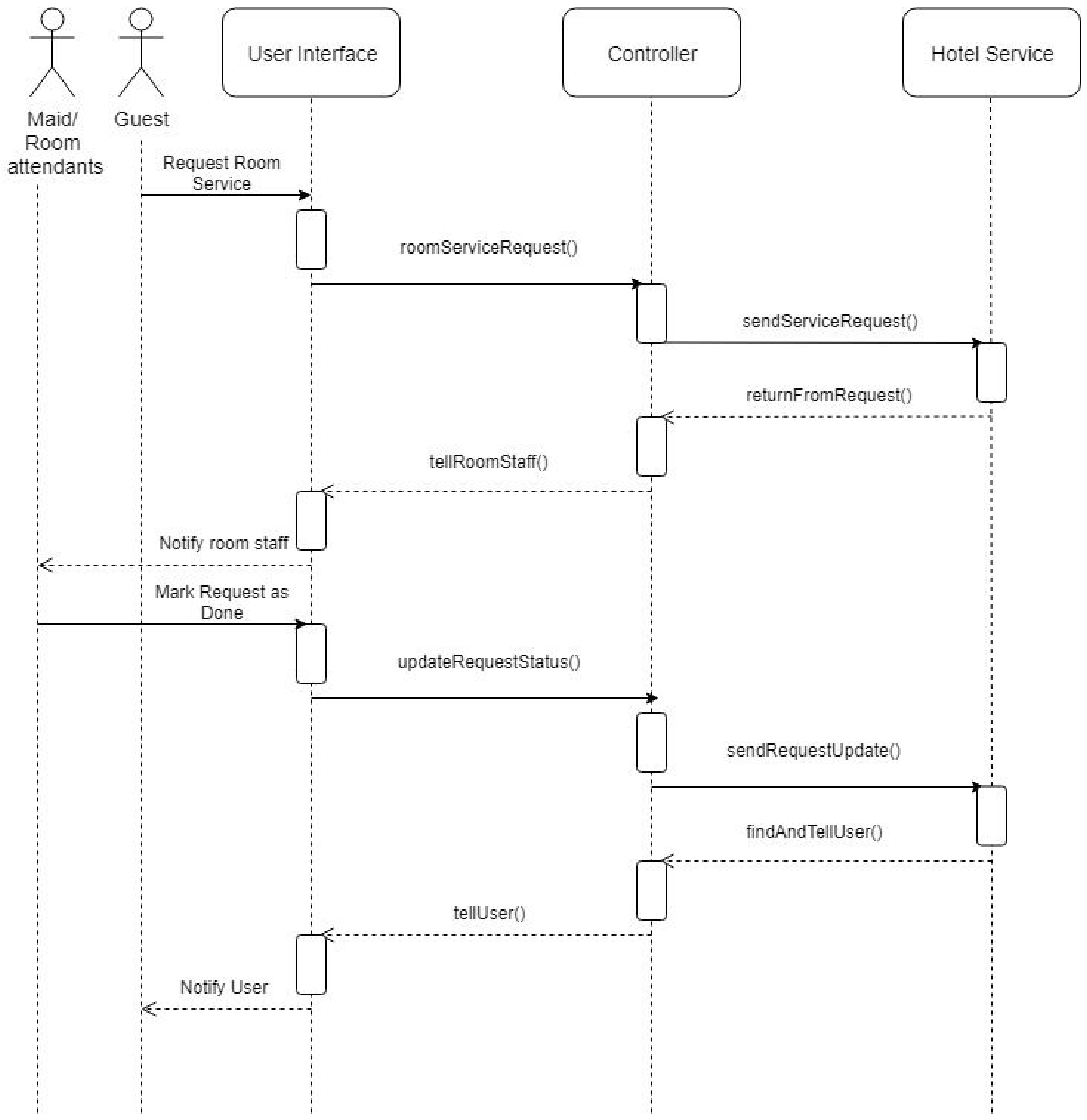
In this use case, the user will be able to make/book a reservation. This process can be roughly broken down into 3 parts. The first part, user will request to make a reservation, prompting the controller to forward the request to Room control. Room control will then send a form to user through controller for filling out. Room control does not need to access database therefore cutting down on over head for this part. Once form is filled out, controller will take form and address the input for room control, so room control can now handle retrieving rooms similar to those requested by guest. Room control will then send that info back to controller so it may display it for user to pick and choose. Once chosen, user will initiate final stage wherein, controller will tell room control to book the request and finalize it through database(Database needs to show that room is booked for however long guest is staying).On fulfillment, database will return confirmation through controller.

Use Case 5 - Check In



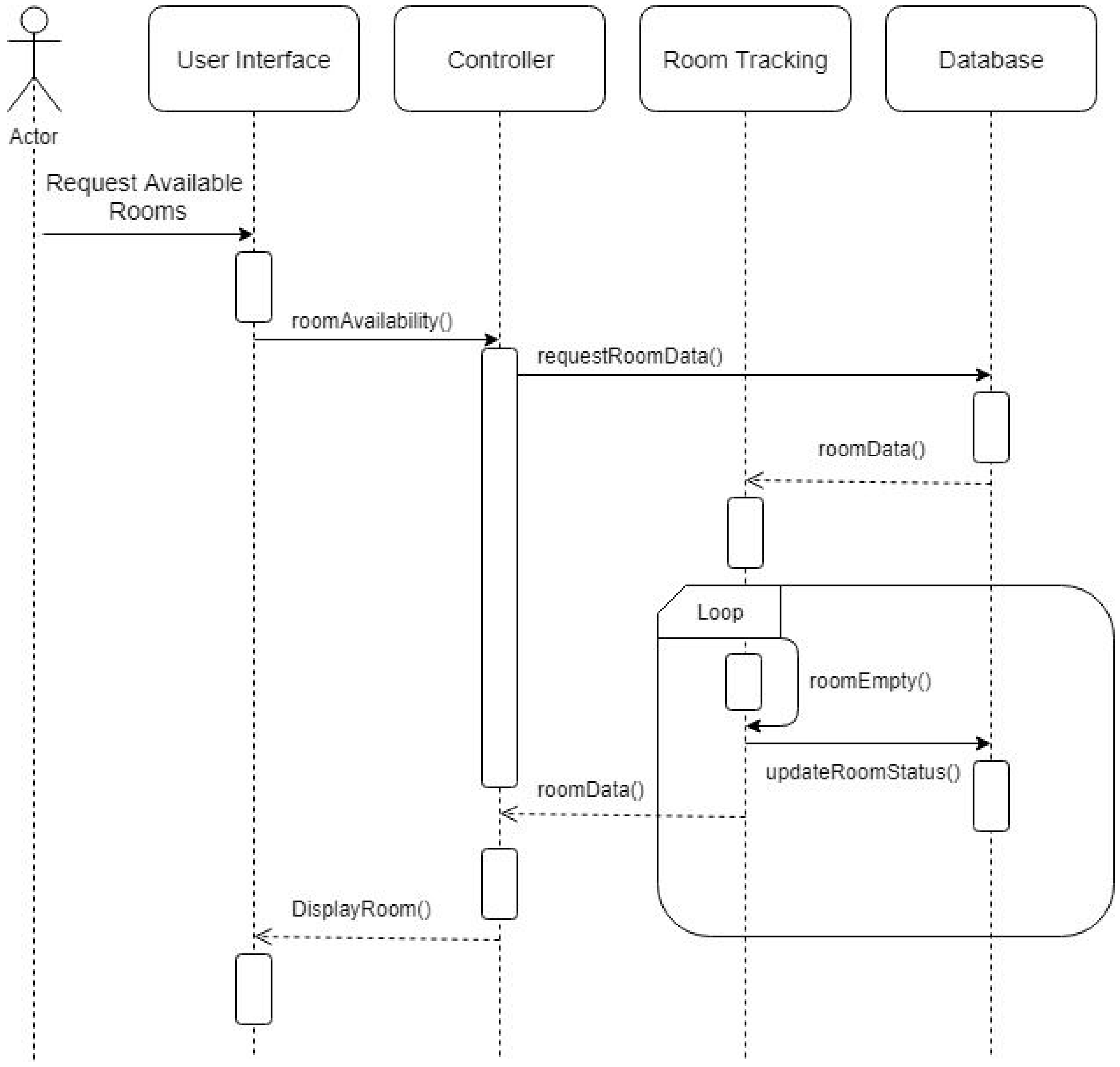
UC-5 refers to how a user can check into their hotel room using our application. It starts off with us assuming user has logged in, clicks check-in button on the user interface. The controller will run the check in function. From here we need to make sure the user has a reservation so the controller will make a request to the database to verify the reservation. The database will confirm the reservation and send it to the room controller so they can provide the user a key for the room and check the user in. After that the control sends the request to display the key on the user interface as well as update the room status in the database.

Use Case 6 - Request Room Service



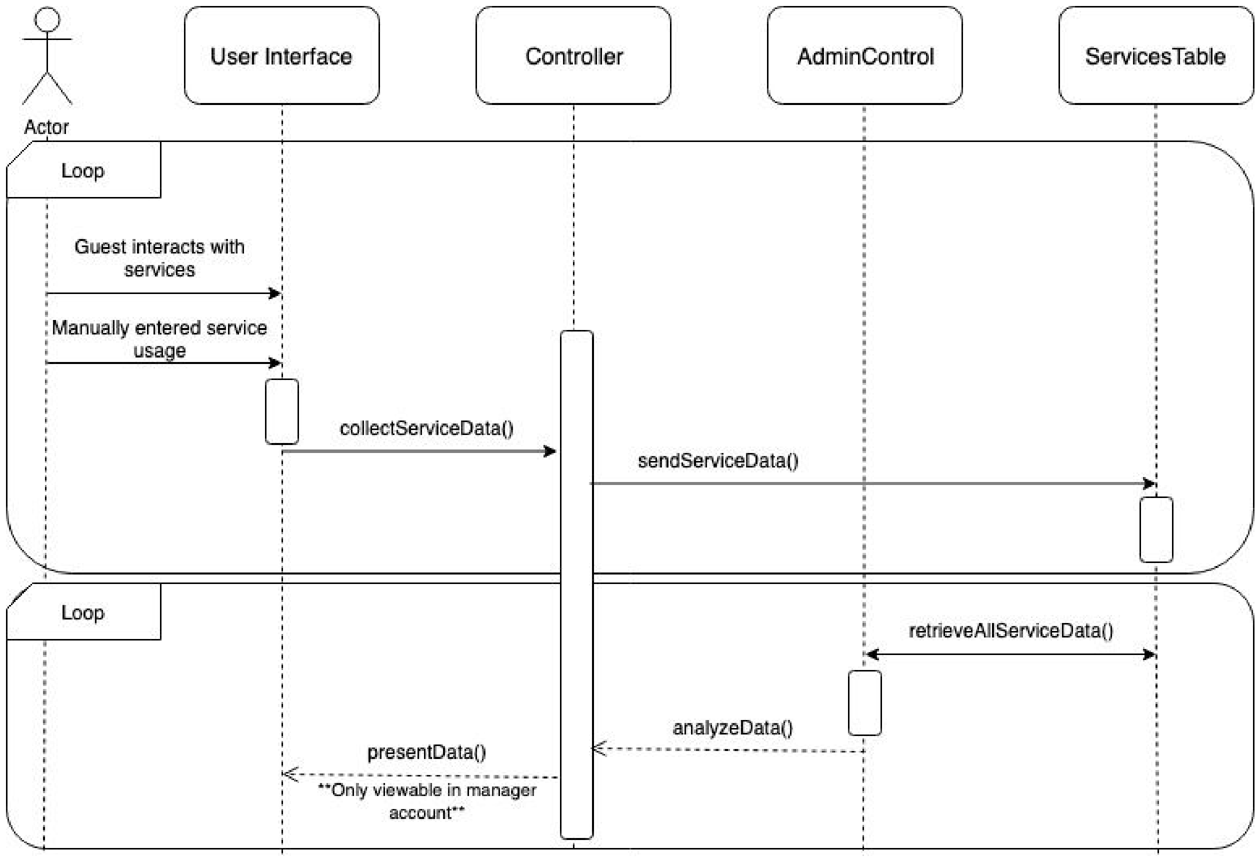
In this use case, the user will be able to call for room service(cleaning, replacing sheets, etc).One of the many functionalities of the app is streamlined requests like this. To access this service, user will input a request from the UI through which controller will receive a signal to Hotel Services which handles these calls. Hotel Services will then forward the request to room attendants to fulfill said request and once request is done, request to mark as done will be passed through controller back to Hotel Services. Once it has been marked as finished, guest will receive a notification of completion as well.

Use Case 7 - Check Availability of Rooms



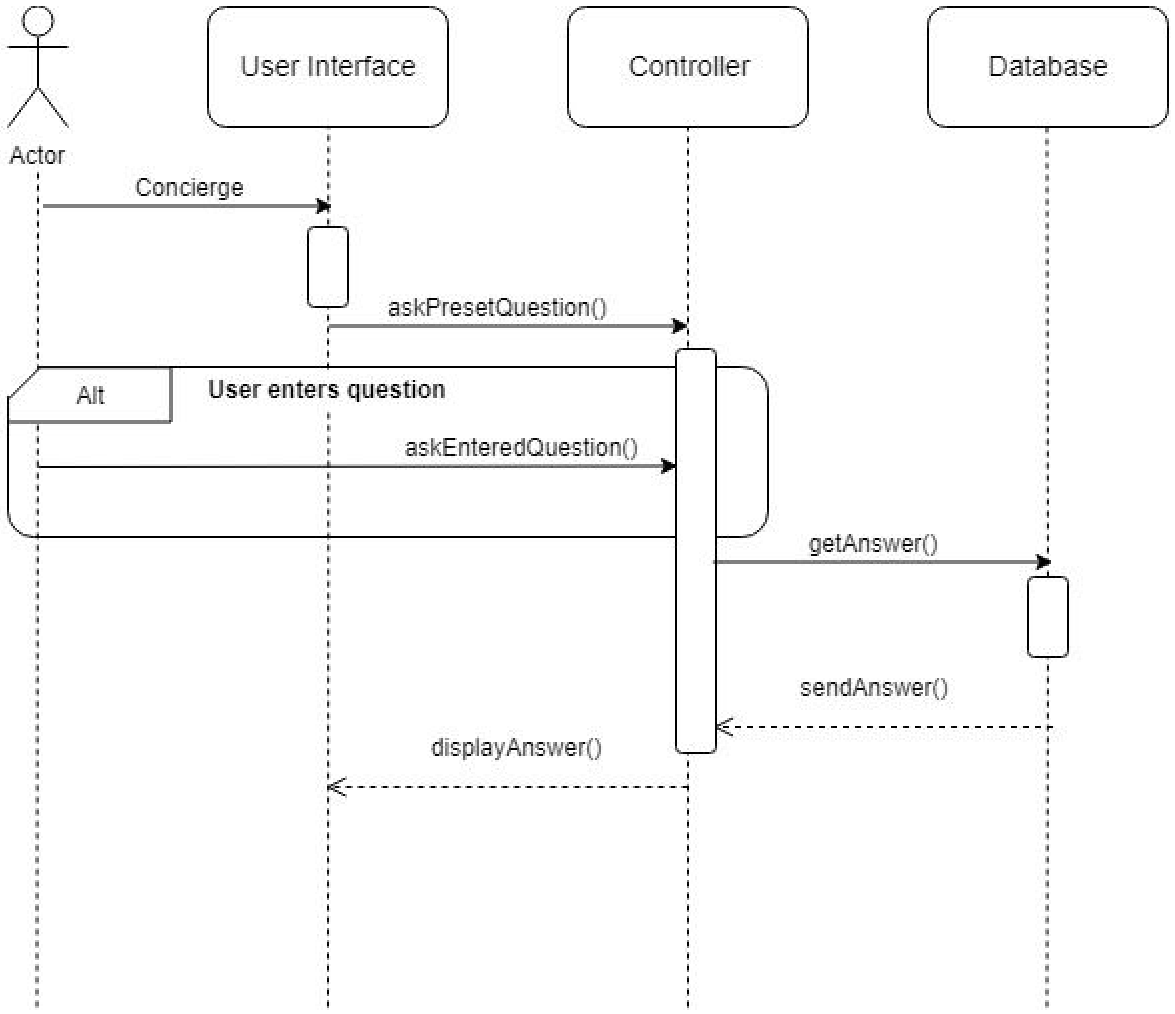
UC-7 allows a user to check the availability of rooms at the hotel. Once they make the request the controller request room data from the database. The database sends this information to the room tracking boundary. Room tracking will run a loop to check if the room is vacant or not. Once the loop runs it will update the status of the room in the database as well as send the data to the controller. The controller then sends the request to display the information on the user interface on whether a room is vacant or not.

Use Case 8 - Review Guest Data



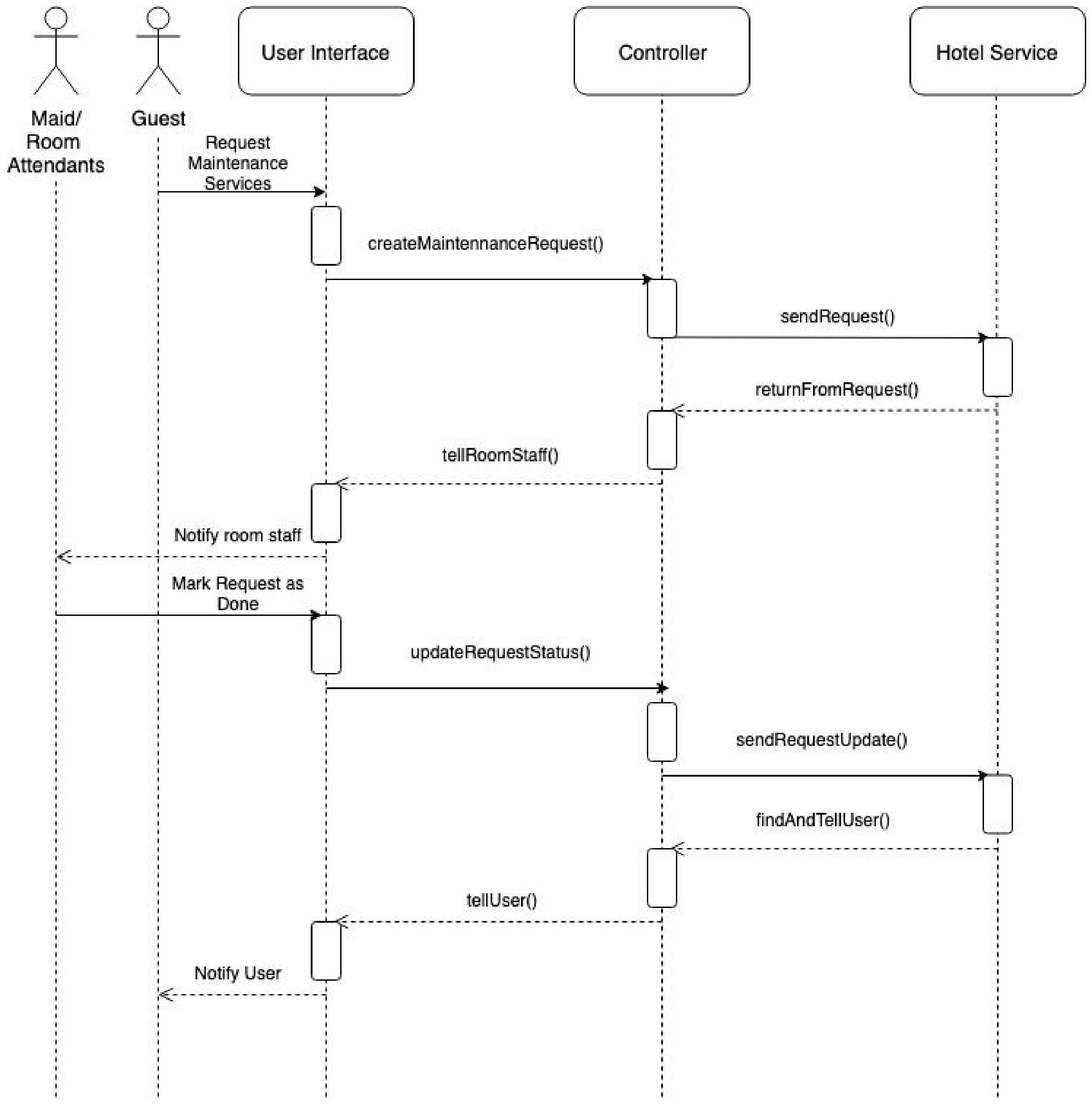
One of the benefits of having an app for hotel automation, is the ability to collect and analyze data. This is a very important use case because it allows administrators to see the usage of the different services provided by the hotel. As guests use the services, or as staff members manually enter usage from guests who do not use the app, the data will be collected by the main controller and sent to the database for storage. Specifically, within the database, we will have a ServicesTable which stores this data along with the frequency and time it was used. The ServicesTable was not in the concepts from the first report because we grouped it together under Database. This process is encapsulated in a loop because as guests continue to use services, the data will be sent and updated in the ServicesTable. The second part of the process involves the AdminControl concept to collect the data from the table. Within AdminControl, we will implement simple data analysis algorithms to be able to present the data back to the administrator. This is also in a loop because as the ServicesTable is updated, the AdminControl must also update the statistics presented.

Use Case 9 - Concierge



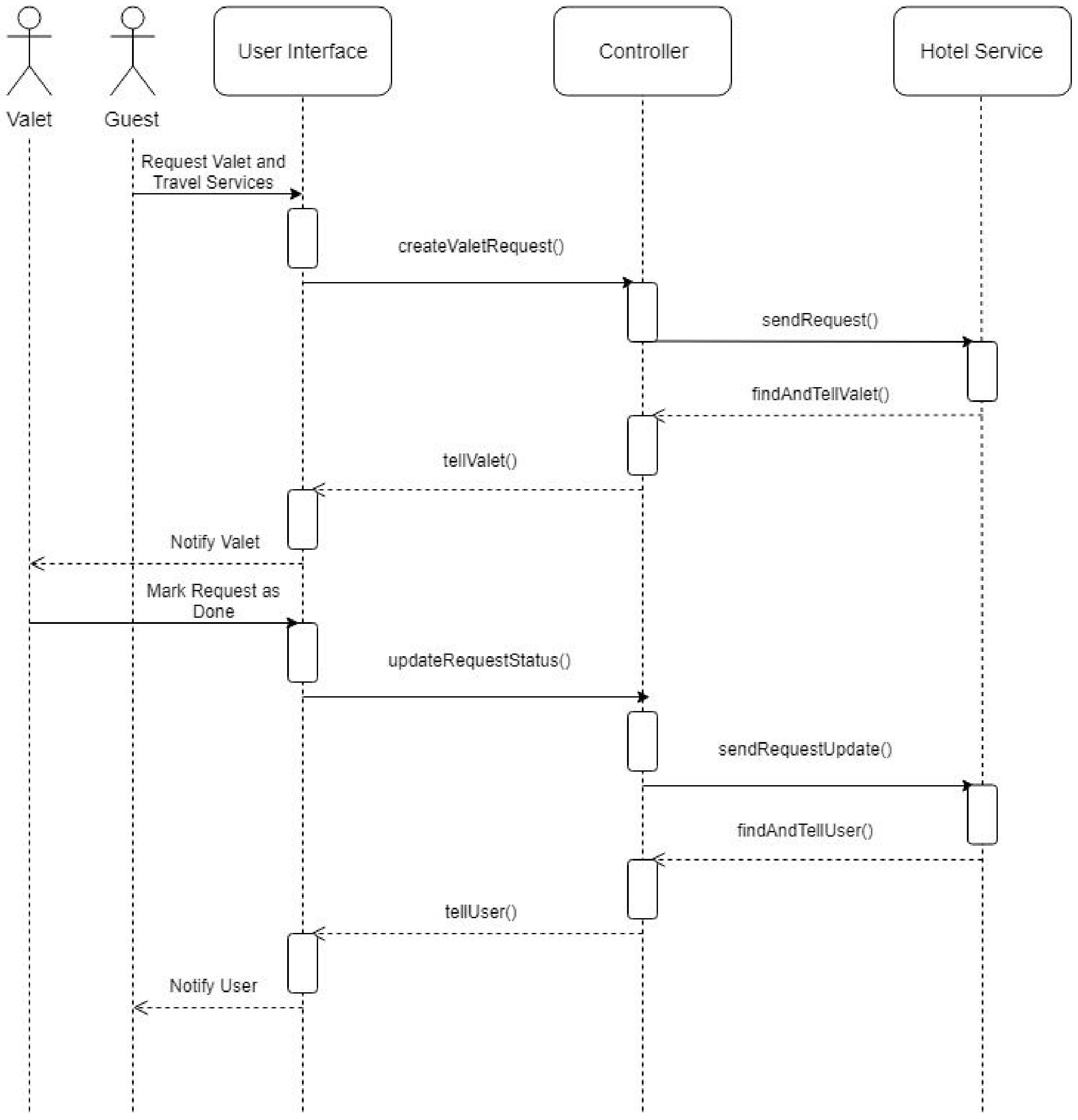
The concierge tab will allow users to utilize an automated concierge service, providing a quick way for users to ask questions without needing to leave their rooms. The process is as follows: the user will first navigate to the concierge tab in the app, then be shown a list of preset frequently asked questions to choose from. Or, if the user cannot find their question in the preset list, they can enter their own question. The question contained in the askPresetQuestion() and askEnteredQuestion() functions are sent to the database, where getAnswer() will interpret the question and search for an answer. Using sendAnswer(), the database returns the information back to the application for the controller to display to the user using displayAnswer(). If getAnswer() cannot find an answer in the database, sendAnswer() will automatically return a message displaying the front desk concierge hours and phone number, directing the user to call or see the front desk later for help.

Use Case 10 - Maintenance Requests



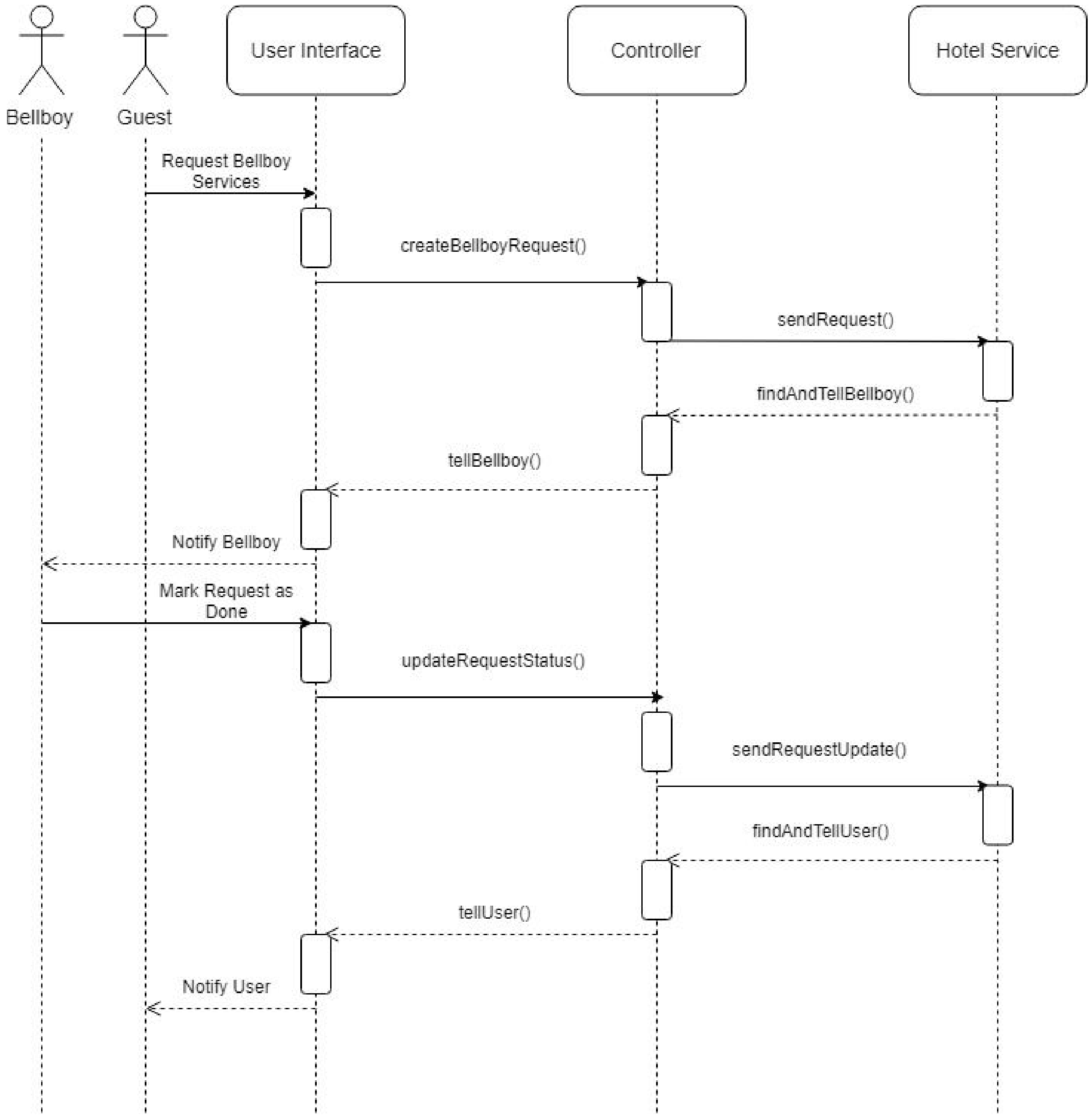
The maintenance requests tab will allow for an efficient way for guests to access request forms for problems within their rooms. Through the app, they will directly be able to navigate to the maintenance request tab, then guests will be able to directly submit their complaint. This is part of the createMaintenanceRequest() function. Consequently, the request will be sent to the database (sendRequest()) and returned to the room staff (tellRoomStaff()) which will allow the room staff to be notified. Once the request has been handled by the room staff and update of the status will be sent through the updateRequestStatus() and then it will be sent to the database. After this, the guest will be informed about the update through the findAndTellUser() and tellUser() functions.

Use Case 11 - Valet & Travel Services



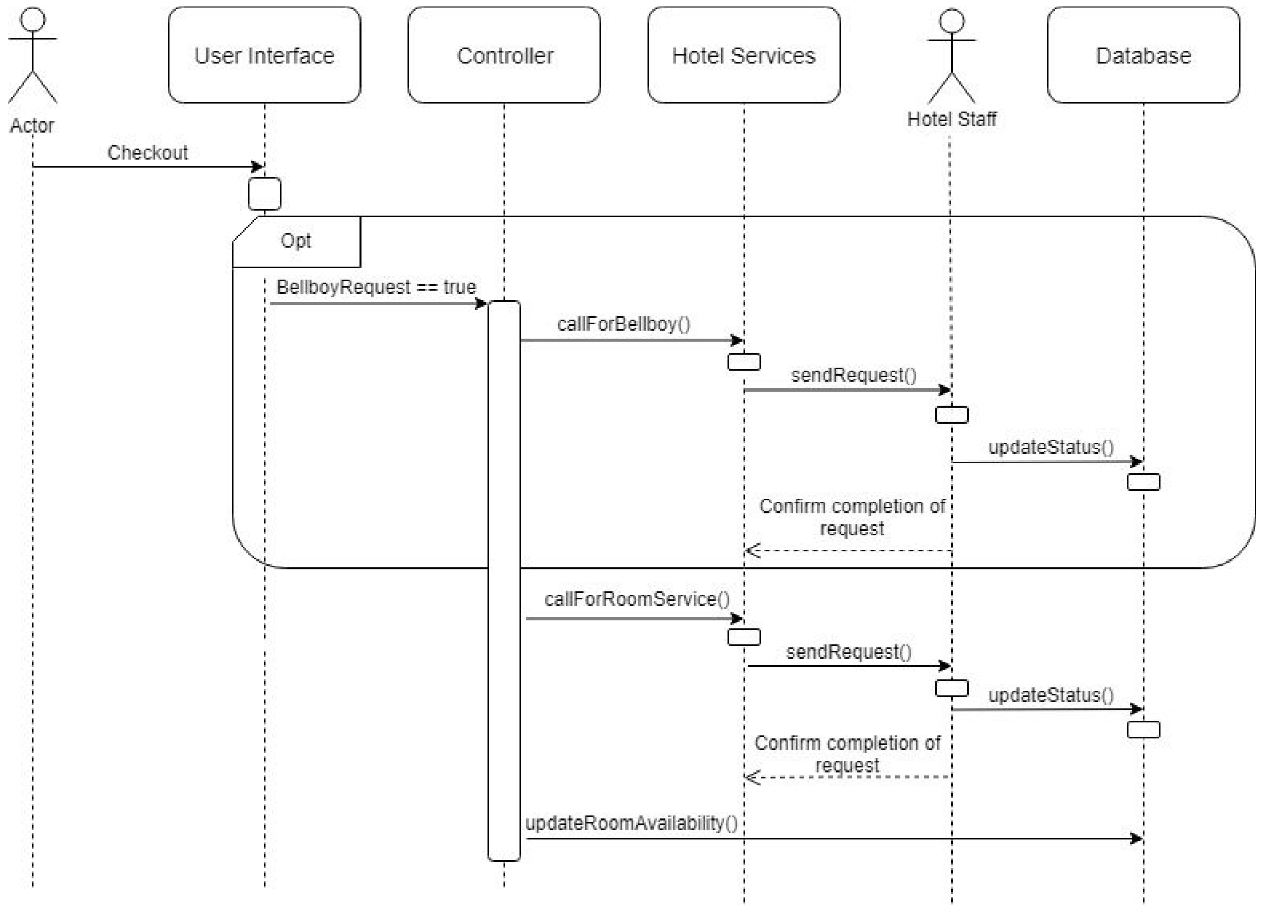
One of the many hotel services includes valet and travel services. This use case is a fundamental service within a hotel. As the guest sends a valet or travel request, the user will specify if they will need to be transported or to have parking services as they submit a request. As staff members, they will receive the guests requests and work to complete the quests requests. They will use the app from the administration side and update the status of the request in the database. The user will receive a confirmation of the complete requested and be notified and will be directed to the home page. This is an outline of how valet and travel services will be processed.

Use Case 12 - Bellboy Requests



A bellboy system was decided upon to make sure that customers could easily get their luggage taken to their room. The guest will tap on their phones to make a request. This request will be sent to the Bellboy in the form of a notification on his/her app. The bellboy can mark this request as done, and this information will eventually be sent to the guest who will receive a notification that their luggage has been moved to their rooms.

Use Case 13 - Checkout

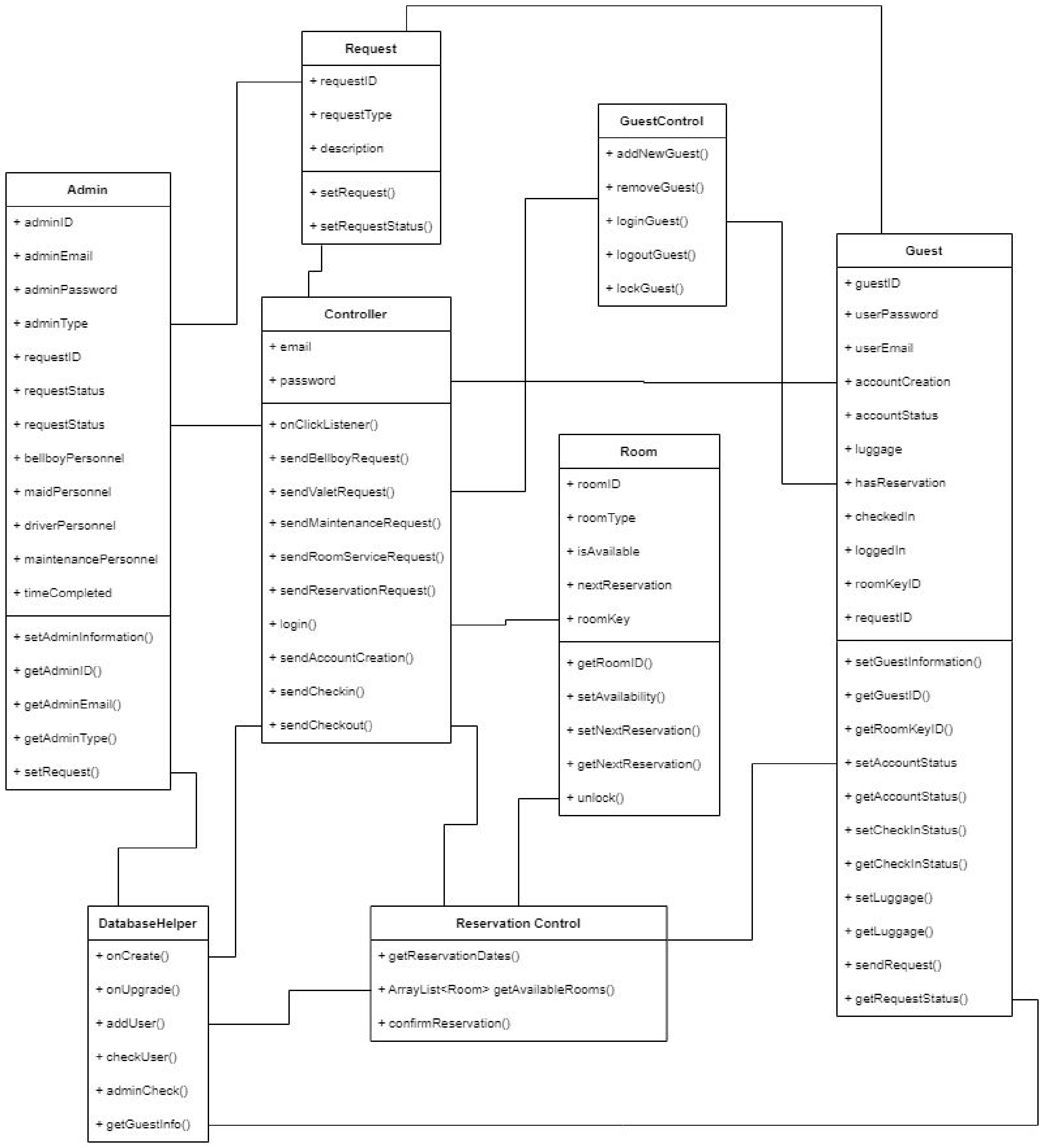


The checkout option through the app is essential to confirm the user has finished their stay at the hotel. After the user has chosen to checkout, they are given the option to request bellboy service to help bring their luggages down to the lobby. On the other hand, room service is automatically notified that they are needed in the room the user has just checked out of. Staff members, bellboys and maids in this case, will use the app to receive their respective requests and to update the statuses of them as well. As soon as the maids have updated the status of this specific request to completed, this will automatically trigger the database to update the number of available rooms.

Chapter 2

Class Diagrams and Interface Specifications

2.1 Class Diagram



2.2 Class Data Types and Operation Signatures

Controller

The controller coordinates actions of all concepts associated with the use cases and logically groups the subsystems to work together. It is the main interface between the system and the application that the user sees.
The controller will be handling all user interactions with the app, such as button clicks and entering text information. It is also responsible for transitioning XML pages as the app is event-controlled and the user can navigate to multiple pages.

Methods:

1) On-click listeners for all buttons
    a) Will trigger different pages in the UI and button clicks may cause any of the below methods to be called
2) sendBellboyRequest()
    a) Will call the Hotel Services Controller
3) sendValetRequest()
    a) Will call the Hotel Services Controller
4) sendMaintenanceRequest()
    a) Will call the Hotel Services Controller
5) sendRoomServiceRequest()
    a) Will call the Hotel Services Controller
6) sendReservationRequest()
    a) Will call the Reservation Controller
7) login(string email, string password)
    a) Will call the Guest Controller and allow access to rest of app
8) sendAccountCreation()
    a) Will call the Guest Controller and allow user to login
9) sendCheckin()
    a) Will call the Guest Controller
10) sendCheckout()
    a) Will call the Guest Controller

Guest

Attributes:

1) int guestId

        a) Unique ID number given to each guest
2) String userPassword
    a) Specific password of guest to login
3) String userEmail
    a) Specific email address of guest to login
4) Date accountCreation
    a) Stores when exactly the account was created
5) String accountStatus
    a) Determines the status of the account (active, inactive, locked etc)
6) int luggage
    a) Determines whether the guest has specified if they have luggage or not
    b) Information used by Bellboy service
7) boolean hasReservation
    a) Boolean value to see if guest has made a reservation for a room, in which case they may check in upon arrival
8) boolean checkedIn
    a) A boolean value to see if guest is checked in or not
    b) Checked-in status confirms that the guest is at the hotel currently
    c) Going from a checked-in equals "true" status to a checked-in equals "false" status indicates that the guest has finished their stay and have now checked out
9) boolean loggedIn
    a) A boolean value to see if the guest is logged into their account
10) String roomKeyID
    a) Integer value with a unique room key ID number for guests to unlock their room doors
11) int requestId
    a) Unique ID number pertaining to a new request a guest has made

Methods:

1) void setGuestInformation(int guestId, String userEmail, String userPassword)
    a) Creates new guest ID
    b) Stores new guest's entered email and password
    c) Creates roomKeyID variable for guest's room
2) String getGuestId()
    a) Get guest ID based on guest information
3) String getRoomKeyID()
    a) Get guest room key information based on guest information
4) void setAccountStatus(String status, boolean loggedIn)
    a) Sets account status of account corresponding to guest ID (active, inactive, locked, etc)

        b) Sets loggedIn variable of account
5) String getAccountStatus()
    a) Gets account status of account corresponding to guest ID (if user is logged in, if account is active, inactive, or locked)
6) void setCheckedInStatus(boolean hasReservation)
    a) Checks hasReservation variable, then sets checkedIn variable
7) Boolean getCheckedInStatus()
    a) Gets check-in status of account corresponding to guest ID
8) void setLuggage(int luggage)
    a) Sets number of luggage pieces user is bringing
9) int getLuggage()
    a) Gets number of luggage pieces user is bringing
10) int sendRequest(Guest guestId)
    a) Creates a new request ID
    b) Sends the request ID to administrative staff containing information of the guest who sent the request
    c) Returns request ID for guest to check on updates pertaining to their request
11) String getRequestStatus(requestId)
    a) Returns the request status of "completed", "in progress", or "processing" based on request ID

<u>GuestControl</u> - Interacts directly with main controller for functionality

Methods

1) addNewGuest()
    a) Add new guest to database
2) removeGuest(int guestId)
    a) Remove a guest from database, specified by ID number
3) loginGuest(int guestId)
    a) Mark boolean value to true if guest is successfully logged in
4) logoutGuest(guestId)
    a) Mark boolean value to false if guest logs out of their account
5) lockGuest(guestId)
    a) Mark boolean value to true if guest has been locked from their account

<u>RequestObject</u>

<u>Attributes:</u>

1) Int requestId
    a) Identifying number of a new request
2) String requestType

    a) String value that contains the type of request a guest has made (“bellboy”, “maintenance”, “room service”, or “travel”)
3) String description
    a) String value containing a description of the guest’s problems or information/specifics of the request

<u>Methods:</u>
1) Void setRequest(String requestType)
    a) Sets the type of request to one of “bellboy”, “maintenance”, “room service”, or “travel”
    b) Sets details of request (i.e. description of problem or specifications)
    c) Creates a new request ID for this request, the number of the request will depend on the type of request, for example, if the user has requested a bellyboy, then the first integer of the request ID will begin with a 1, if maintenance, then the first integer of the request ID will begin with a 2, etc.
2) void setRequestStatus(requestId)
    a) Sets status of request to “completed”, “in progress”, or “processing”

Admin

<u>Attributes:</u>
1) int adminId
    a) Unique ID number given to a member of the administrative staff
2) String adminEmail
    a) Email of an administrative staff to login
3) String adminPassword
    a) Specific password of staff to login
4) String adminType
    a) Specify the exact role of Admin
    b) “Bellboy”, “Maid”, “Manager”, “Driver”, “Maintenance”
5) int requestId
    a) Contains request Id
6) String requestStatus
    a) Contains status of request as “processing”, “in progress”, or “completed”
7) int bellboyPersonnel
    a) Number of available bellboys to perform bellboy requests
        i) If 0 requestStatus remains “processing”
        ii) If >0 requestStatus becomes “in progress”
8) int maidPersonnel
    a) Number of available maids to perform room service requests

        i) If 0 requestStatus remains “processing”
        ii) If >0 requestStatus becomes “in progress”
9) int driverPersonnel
    a) Number of available valet drivers to perform valet & travel requests
        i) If 0 requestStatus remains “processing”
        ii) If >0 requestStatus becomes “in progress”
10) int maintenencePersonnel
    a) Number of available members of maintenance to perform maintenance requests
        i) If 0 request remains “processing”
        ii) If >0 requests becomes “in progress”
11) int timeCompleted
    a) Contains time needed for a request to be completed. A staff member who has received a request will have a timeCompleted integer assigned to them. A clock will determine if the value stored in timeCompleted since the request started has been reached, in which case timeCompleted will be set to 0 and the available personnel variable associated for the type of staff member will be incremented indicating that the staff member has completed their task and is now available for another. The request status will also then be updated to “completed”

Methods:

1) void setAdminInformation(String adminEmail, String adminPassword, String adminType)
    a) Set admin entered email
    b) Set admin entered password
    c) Set admin entered type
    d) Create admin’s new ID
2) int getAdminId()
    a) Gets admin’s ID
3) String get adminEmail()
    a) Gets admin’s email
4) String getAdminType()
    a) Get admin’s type
5) String setRequest(requestId)
    a) Checks number of available personnel depending on who the request was made to
    b) Sets status of “processing”, “in progress”, or “completed” depending on the available personnel variable values. If no personnel are available, this method will be put in a loop to check every minute if a staff member has become available. If one has become available, sets status from “processing” to “in progress”

<u>Room Object</u>

<u>Attributes:</u>

1) int roomId
   a) Unique number assigned to each room
2) String roomType
   a) “Single”, “Double”, “Queen”, or “King”
3) boolean isAvailable
   a) Flag to indicate if the room is occupied or not
4) Date nextReservation
   a) Date object to indicate when the room is reserved if it is not currently in use
5) String roomKey
   a) Unique room key string given to each guest staying in a specific room, or to staff to enter the room

<u>Methods:</u>

1) int getRoomId()
   a) Returns the unique number of the room
2) void setAvailability(boolean isAvailable)
   a) Method to set the availability to of the room to true or false
3) void setNextReservation(Date nextReservation)
   a) Method to set the reservation of the room if a new request was made successfully
4) Date getNextReservation()
   a) Method to get the next reservation date of the room
5) boolean unlock(int roomKey)
   a) Method to compare if user entered room key matches with the room’s actual key value, will return true if unlocked and false if the key do not match, leaving the door locked

<u>ReservationControl</u> - Implements functionality to create new reservations

Methods

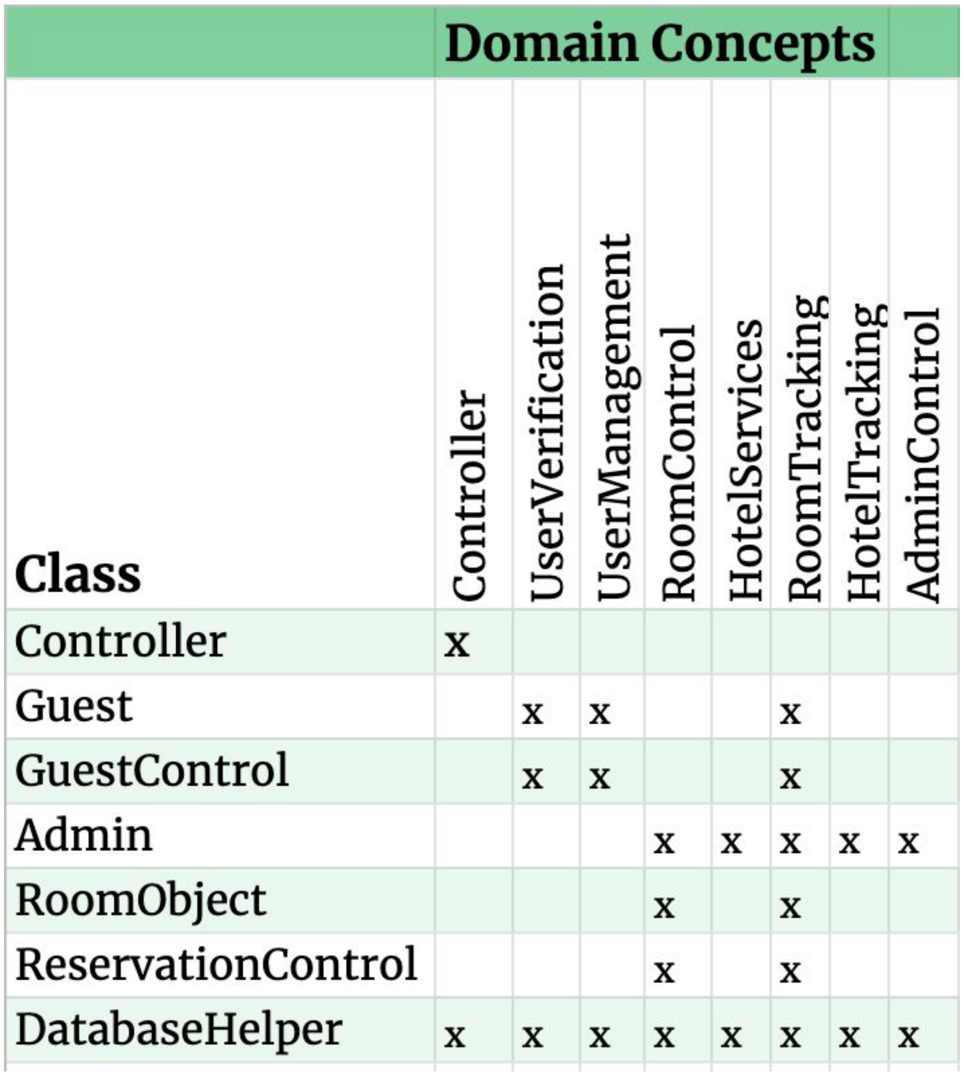
1) getReservationDates()
   a) Gets information entered by guest in the UI
2) ArrayList<Room> getAvailableRooms(Date startDate, Date endDate)
   a) Based on the duration of stay, check all the rooms that are available and return an ArrayList of these Room objects
   b) The controller will output this ArrayList to the UI so the guest can see the available rooms
3) confirmReservation(Room bookedRoom)
   a) When the guest selects a room, send that object back to mark the room as reserved for those dates

<u>DatabaseHelper</u>
Main object that manages the database objects and all the methods that directly input and extract data from tables. As we continue implementing functionality, the DatabaseHelper methods will continue to increase. Maintaining database information is the backbone of our app because all the data is used to help serve the guest.

<u>Methods</u>

1) onCreate()
    a) Default by Android studio
2) onUpgrade()
    a) Default by Android studio
3) addUser()
    a) Adds a new user to the table
4) checkUser()
    a) Checks if a user exists in the table
5) adminCheck()
    a) Checks if an admin login is valid.
6) getGuestInfo(int guestId)
    a) Returns a Guest object from the given ID number

2.3 Traceability Matrix



	Domain Concepts							
Class	Controller	UserVerification	UserManagement	RoomControl	HotelServices	RoomTracking	HotelTracking	AdminControl
Controller	x							
Guest		x	x			x		
GuestControl		x	x			x		
Admin				x	x	x	x	x
RoomObject				x		x		
ReservationControl				x		x		
DatabaseHelper	x	x	x	x	x	x	x	x

The **Controller** class will be the interconnected with all other classes. This is the only domain concept that's itself because everything is directly accessible through this class.

The **Guest** object will keep track guest account information on luggage status, reservations, check in/out, and account status. This is related to UserVerification because it is needed to check with the database if the email has already been used before. It is also part of the UserManagement concept because all their account information and object fields will be stored in the database.

The **GuestControl** will be responsible for checking proper login request for each guest and marking them as currently logged in and out. Additionally, GuestControl will also be in charge of adding and removing guests from the database.
The **Admin** object will be directly from the domain concepts AdminControl but also interacts with other concepts. This will be the platform in which hotel admin will be able to login to their own system. This falls under the AdminControl as the administrators/hotel staff (i.e bellboy, valet, concierge) will all be the same object, but differentiated by their own separate duties/roles within the system. This distinction between guest and administrator will be necessary for functions within the app.

The **RoomObject** will be related to RoomTracking, and RoomControl. Each Room will be its own object once the guest checks into the room. The object will be used to manage the different parts of the room, such as room key, maintenance request and reversed dates. RoomTracking concept is meant to analyze all the types of rooms available on specific dates, so this will use the Room object data to accomplish this. And the RoomControl concept allows the guest to make a reservation and check-in, so the Room object data will be updated accordingly.

The **ReservationControl** is for retrieving information of reservations from guests and identifying the available rooms for available days. This interacts with the RoomControl and RoomTracking to determine the available rooms at the time of the request. Once the reservation is confirmed, the data is sent back and updated in the concepts.

The **DatabaseHelper** is a separate object to handle all the SQL statements and directly change the stored data. Within this class, there will be methods for each object to interact with, for example, having an addUser() methods to run SQL that add a user to the UserTable. Therefore, the DatabaseHelper will interact with each concept because each concept requires data management.

Chapter 3

System Architecture

3.1 Architectural Styles

Communication

The basis for our RUstaying app lies upon the many functionalities that we provide to our guests through hotel automation. Through Service-Oriented Architecture we are building each of these functions independent of each other. These implemented services communicate messages with our app through our formally defined XML schemas in Android Studio. This allows us to reuse code and simplify the management of said code. SOA helps keep our code well-defined and self-contained with its various functions because at the end, we need robust capability for the app to be able to communicate with an end user and other entities like databases or another user with admin privileges. An advantage of using this architecture is that if we were to scale up or scale down on functions, the rest of the functions would still be just as efficient since they are self-reliant.

Deployment

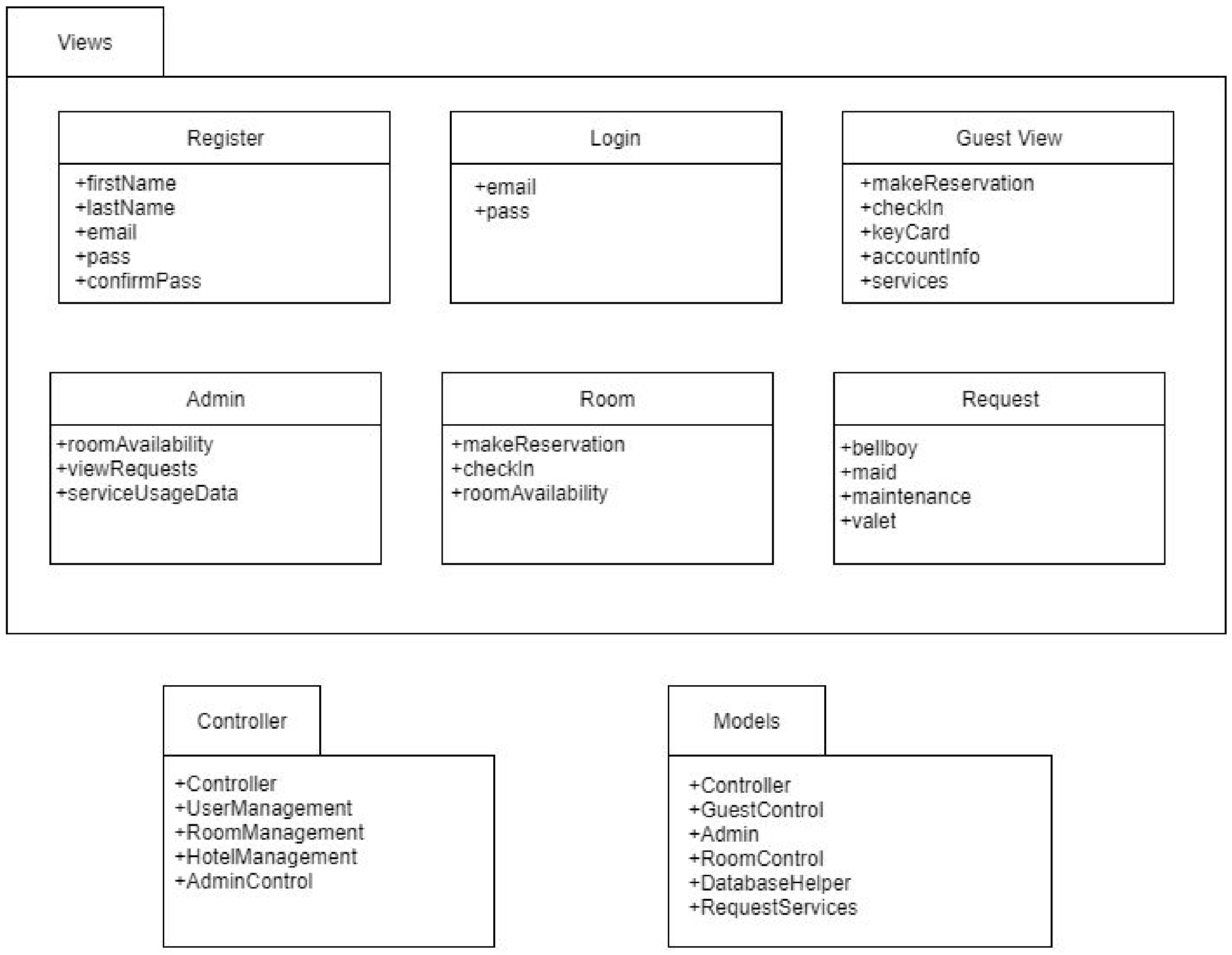
Current version of app uses SQLite. For the deployment of our app, once completely implemented, ideally we would need a database which is accessible through a cloud for when the Model needs to load/store data. This type of interface which is essentially a client-service communication so as a result we are using a client-server model. When there are multiple platforms supporting the app, such functionality for the interface needs to be consistent across all devices so having this model gives us the most efficiency. Ultimately, we rely on these user-server requests and this model gives us the most efficiency and scalability

Structure

Using MVC(Model-View-Controller) for Object Oriented Architecture
In our case, the controller component is what initiates a function, manipulates data in the Model, and returns some values and displays the data for end user to view. Generically, once the user has some input in the UI, the controller will handle the logic and call the function requested. The Model, which in this case are the independent functions(may or may not use database) will load/store data. Here is where all the classes which the RUstaying team has written will come into play. Once controller has received confirmation that the function has completed

partially/completed (app might need more input from UI for complex functions), controller will ask View to update for further instructions.

3.2 Identifying Subsystems



3.3 Persistent Data Storage

In order to keep track of data for long periods of time, we must use persistent data storage. Our app uses Firebase to store data obtained from the user. Firebase is a NoSQL Cloud database management system. Data is synced in real time across all clients, and remains available even when the app goes offline. Data is stored as a JSON and synchronized in real time to every connected client. When adding data to the JSON tree, it becomes a node in the existing JSON structure with an associated key.
To store persistent data, we will be creating objects with key values to have and use these objects to pass data into the JSON tree. We will have objects such as Guest, Room, and Services. Here is short description of each object and its use of persistent data:

```
Public Class 'User'(
        'FirstName' TEXT,
        'LastName' TEXT,
        'guestID' INT,
        'userPassword' TEXT,
        'userEmail' TEXT,
        'accountCreation' DATE,
        'accountStatus' TEXT,
        'hasReservation' BOOLEAN,
        'checkedIn' BOOLEAN,
        'loggedIn' BOOLEAN,
        'roomKeyID' TEXT
)
```

Description: As guestID, userPassword, and userEmail is required to login, this information must be persistent. The system should also keep track of the user first and last name. accountCreation, accountStatus, hasReservation, checkIn, loggedIn are also all important fields to keep persistent as some of these booleans need to be TRUE for the user to activate certain services. roomKeyID should most definitely be persistent as the user would not be able to enter the room without it each time.

```
Public Class 'Room'(
        'roomId' INT,
        'roomType' TEXT,
        'isAvailable' BOOLEAN,
        'nextReservation' DATE,
        ‘roomService’ TEXT,
        ‘maintenanceRequest’ TEXT
)
```
Description: To keep track of each room and its availability, it is vital that we keep fields such as roomId, roomType, isAvailable and nextReservation persistent. Without this persistent data, the staff would not know which rooms are currently available or will be in the future.

```
Public Class ‘Admin’(
        ‘adminId’ INT,
        ‘adminEmail’ TEXT,
        ‘adminPassword’ TEXT,
        ‘adminType’ TEXT
)
```
Description: Since the staff runs the hotel, there is user information available to them. To make sure only an admin can access such information, adminId, adminEmail, adminPassword are kept persistent fields. No one other than a respective adminType should be able to access certain data and services.

3.4 Global Control Flow

Execution Order

This app is procedure-driven up until after registration and login, meaning every user will go through the same steps every time. The rest following is event-driven which means the user determines what the app is used for and which features they want to use. Upon opening the app, all users must either log in or if they do not have an account, register and then log in. Each user can use the app whenever they want and in order of whatever features they would like after being logged in. For example, once the user is logged in, they can choose to do anything like book a room, request a service, or even logout.

Time Dependency

This app is a real-time dependent system that refreshes its information every 5 seconds so that the user and admin can both have the most updated version on room vacancies, check outs, check ins and if a maintenance/room service request has been completed. Every function in the app is either dependent on what the users or hotel staff is doing at any given moment. For example, a guest will check into a room for a set amount of days in real time and hotel staff will be completing services such as room service in real time and the app will be updated.

Concurrency

Our app will not be using multithreading or multiprocessing. The user interface is event-controlled and each action only requires linear execution. There is no need for concurrency because the user can only initiate one action at a time. Synchronization is not directly used within our app but will be used by the database to store information properly. Because multiple users can be using the app at the same time, there could be multiple requests to get or update the data stored in the database. This, however, is managed directly by SQLite in its implementation. We will not need to worry about synchronization because SQLite will be managing concurrent data requests at the same time to ensure the data is not corrupted.

3.5 Hardware Requirements

As a whole, the majority of computational resources are used on the server-side with the use of Android Studio, SQLite database, and a Java backend which can be accessed by Android phones. The system replies on the use of Internet Connection in order to constantly pull data and verify the features of the application the user chooses to the admin and the employees of the hotel. This is also necessary in order to maintain consistency of uses for all of the users and maintain a constant updated version of room status, service status, check out/check in, etc. The internet is also required for the concierge feature of the app in order to help the user find nearby activities/restaurants or allow any questions of the user to be sent to a concierge themselves to answer. Due to the fact that we are using Android Studio, we are able to customize the app's layout, allowing us to be flexible with the resolution, scaling size and disk space. Android Studio allows us to create alternate layouts as well in order to adjust to how the user is holding their phone while viewing the app- horizontally or vertically. Users will be able to access the app through most Android smartphones or simulators that act as one.

Chapter 4

Algorithms and Data Structures

Algorithms

The main functionality of RUStaying revolves around adding new guests' data to our database, updating existing guests' data in our database, keeping track of room availability from our database, keeping track and carrying out hotel services from our database, and using simple one step mathematical operations such as addition or subtraction to keep track of available employees. As a result, our application has no real algorithms needed for any calculations or predictions.

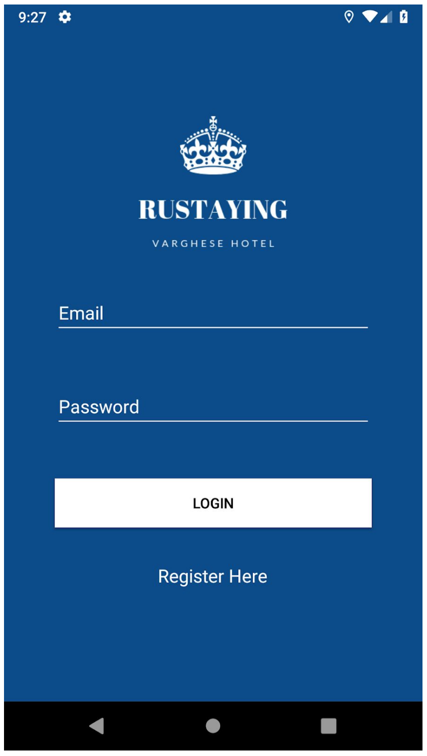
Data Structures

The place where we use complex data structures is within the data storage. We upload the data about all the Users of the application into a Google Firebase. Firebase is a NoSQL database and stores its data in JSON format. The database is essentially one large dictionary object (similar to a Hash Table) with various different dictionary objects nested within it. Thus, whenever we need to access we need to pull up information regarding a specific user we can use a key value, which would be a userID, to get the value, which would be all the information about that user stored in another dictionary. You would access the parts of the user information dictionary just as you would access the previous dictionary of all the users. We had to work with this dictionary structure due to the way Google firebase operates. Since Firebase is very easy to integrate into Android applications, We decided it would make sense to use within our app. Thus we had to think about how we would represent our users within a dictionary. Thus, it was helpful to think of our users as objects with attributes that could be represented with the dictionary.

Chapter 5

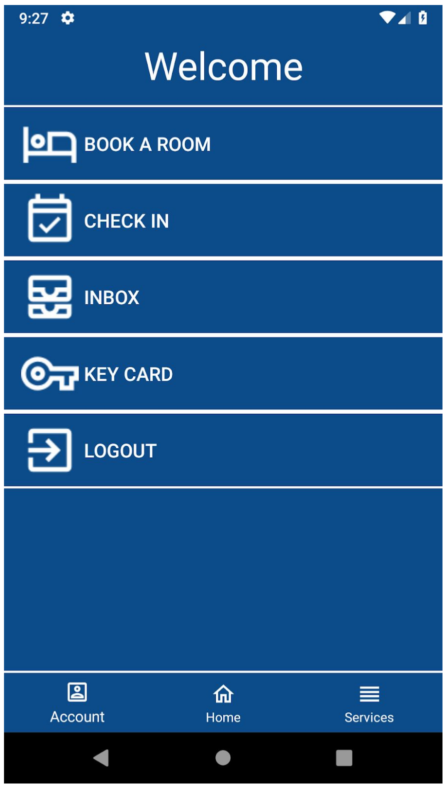
User Interface Design and Implementation

Implementation of our app was built on Android Studio using Activities to display the contents of each page that the user sees. We use XML to create the contents that the user sees in each activity and Java to provide functionality for all the elements on the screen such as textboxes and buttons. The application can be run locally on an emulator built into Android Studio or you can connect an Android device and run our app. The interface is easy to use and displays information in an easy to read fashion. Every button is labeled and provides users with messages using Android Toast Widget to provide feedback when a user has completed any action successfully. The UI design is simple and on focuses on the necessary information. There are no pictures or unnecessary data to distract the user from the task they want to accomplish. Overall the design from our mockups is unchanged. We added some stylistic changes by using Material Design by Google and changing the colors, However the overall frame and design is simple and functional. Below we have a couple pages of our application:

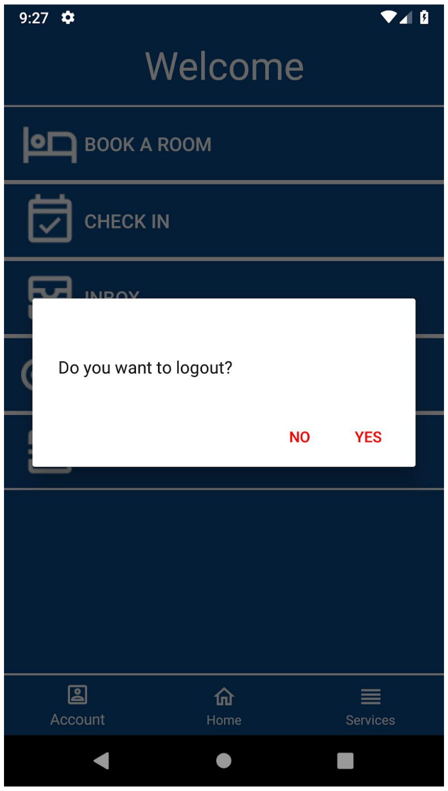


This is the main page that opens up when a user opens the app. As you can see we made it very simple with a small logo, the name of our app, and large text fields and buttons so it is very easy for the user to see.

The register page is also very simple with every box fully labeled so the user knows exactly what to input. We also have functionality in our code that provides popup notifications when text fields are empty so the user knows if they are doing something wrong.



After logging in the user is brought to the main page. From here the user can access all the things the app provides. The buttons are all large, clearly labeled, and have images to help the user.



Do you want to logout?

We also have alert dialogs in case a user accidentally clicks a button. This is all to make sure our app is easy to use and the user is correctly deciding each action they want to do.

Chapter 6

Design of Tests

Test Case 1: Guest Login Use Cases: UC-1, UC-2 Pass/Fail: A successful test will verify the guest login information to allow or deny access Function Called: login(guestEmail, guestPassword)	
Test Procedure:	Expected Results:
Guest enters their correct email and password	The system verifies with the database and allows access to the guest
Guest enters incorrect email or password	The system notifies guest of incorrect login and prompts to login again

Test Case 2: Guest Registration Use Cases: UC-1 Pass/Fail: A successful test will add a new guest into the system Function Called: addUser(guestEmail, guestPassword)	
Test Procedure:	Expected Results:
Guest fills in all required fields of registration form	The system creates new account and stores it in the database
Guest already has an account that exists	The system notifies guest that their email is already in use by another account
Guest does not fill in all the required fields	The system notifies guest to fill in required fields

Test Case 3: Admin Login Use Cases: UC-1, UC-2 Pass/Fail: A successful test will allow an admin to login with preset email address and password credentials Function Called: adminLogin(adminLogin, adminPassword)	
Test Procedure:	Expected Results:

Admin enters their correct email and password	The system verifies with the database and allows access to the admin
Admin enters incorrect email or password	The system notifies guest of incorrect login and prompts to login again

Test Case 4: Feedback Form Use Cases: UC-13 Pass/Fail: The test passes if the feedback data from the guest is properly sent and stored into the database Function Called: collectFeedbackData()	
Test Procedure:	Expected Results:
The guest fills out feedback form and data is sent to the database	System updates the database with the new data received

Test Case 5: Design Admin Pages Non-functional Test Function Requirement: REQ-18 Pass/Fail: The test is successful if the proper pages are displayed when admin logs in	
Test Procedure:	Expected Results:
Admin has logged in	The system noticies admin login and redirects to proper XML pages

Test Case 6: Display All Rooms Use Cases: UC-4, UC-7 Pass/Fail: The test passes if all the rooms available in the hotel are presented to the guest when they go to create a reservation Function Called: populateAllRooms()	
Test Procedure:	Expected Results:
Guest requests new reservation	System will populate the UI elements to show all the rooms available

Test Case 7: Filter Available Rooms Use Cases: UC-4, UC-7 Pass/Fail: The test passes if all the rooms unavailable in the hotel are not displayed to the guest while making a reservation

Function Called: filerRooms(roomType, startDate, endDate)	
Test Procedure:	Expected Results:
Guest requests new reservation and specifies room type and duration of stay	System will update the list of all available rooms to match the requirements of the guest

6.2 Unit Testing

Unit testing is software testing for each unit of a software. Unit is the smallest part of the program. For our case, the smallest component of the software is a method. Each method has one or more input and a single output. For unit testing, we decided to test six integral methods of the application. You will find those methods in the test cases listed above. Our goal is to validate that each unit or method performs as designed. For example, we are testing Guest Registration (Test case 2) to check that each user can successfully create an account. If this method does not properly perform its tasks, the user will not be able to use any further features of the app. By providing such testing, we can ensure that our code is reliable and reusable.

6.3 Test Coverage

Test coverage is the amount of testing done on a piece of software by a set of tests. It includes gathering information about parts of the code you are going to test. So, it ensures that the tests you are running are successfully testing what they are designed to test. Benefits of test coverage are that it guarantees the quality of your test and also helps identify parts of your code that are fixed or altered. It helps to create a number of various test cases to increase coverage. As mentioned under Unit Testing (3.1), we want to guarantee that our users can successfully create their accounts. Without so, they would not have further access to the app. We are providing a diverse set of test cases to ensure that the method works properly for different sets of input. By increasing the number of diverse test cases we perform, we are increasing test coverage.

6.4 Integration Testing

Integration testing is one level above unit testing that tests a group of units together. The point of such testing is to ensure units work well together and eliminate any faults in interaction of units. For example, when we test User Login, we check the interaction of many units coming together. We are directly testing the methods required to have a user login, as well as algorithms to store and receive user information required to login. We test the database to make sure information the user typed in is valid. When we test the Feedback Form, we are again testing its direct code as well as the database. When a guest fills out the form, we want to ensure that the data is being properly stored in the database. The system would update the database when the new information has arrived. By using integration testing, we are verifying that our methods work together to provide a functional application.

6.5 Algorithm Testing

As mentioned under 1.1 (Algorithms), our application has no real algorithms needed for any calculations or predictions. For testing non-functional requirements, we are testing REQ-18 from Report 1. The requirement states that the app should have a simple and responsive user interface. Our Test Case 5 is a non-functional test that determines if proper pages are displayed when admin logs in. The system is expected to recognize that an admin, not a guest, logged in and should redirect to the proper XML pages.

Chapter 7

Project Management and Plan of Work

7.1 Merging Contributions

As we have now submitted our Full Report 1 and have gotten some feedback on it, we have realized that there are plenty of various areas we needed to improve on. A few of these tasks that needed improvement included delegating tasks, report flow, organization, and consistency. Originally, we would just assign different parts of the report to different subgroups (2-3 max) to complete and then just merge everyone's individual work. This caused a lack of flow in the reports and sometimes we even forgot to check if we hit every point of requirements. It was easy for each individual members to just focus on the work they were assigned, but it is more beneficial as for the group if each member were updated on the general requirements needed for every report so we can also check each other's work. So now, not only, do we conscientiously make efforts to improve we also have at least one person to check if everything written also relates to other parts of the report before submission. There is also a final check with formatting, consistency, and appearance of every report, as well as a table of contents to help organize and guide readers to specific topics addressed amongst many within the report.

7.2 Project Coordination and Progress Report

Below is a detailed breakdown of the various software development parts of the project and how the work is split up. We defined the subgroups and which parts each team is going to develop. This breakdown is a good estimate of the development work we will have in time for Demo 1.

Subgroups based on basic app features:

Subgroup	Task	Description
Keya Patel, Zain Sayed	Feedback form	Design the page and the database table to store the information. Be able to save the information properly when user fills out the form (we'll implement the feature where it prompts the user to fill it out the form when other milestones are complete) (Milestone 8)
Shilp Shah, Mathew Varghese	Setup Room object and create database table of all available rooms	Have all proper fields in database table. And properly think about the amount and types of rooms the hotel will have. (Milestone 5)
Purna Haque, Mohammed Sapin, Nga Man (Mandy) Cheng	Make preset emails and passwords to indicate an Admin login	The admin XML pages should be displayed if the login user is determined to be an admin. (Milestone 4)
Eric Zhang, Rameen Masood, Thomas Tran	Bellboy Service	Create pages for guest to request this service. Implement a staff notification when new service is requested and user notification when request is

		accepted(Milestone 10)

As stated earlier in the report, while we were working on our assigned tasks for development, we realized how inefficient it is to utilize the data between different clients using SQlite as the data is only saved locally. We realized we would have needed to implement a server where we can remotely access the database to use between different user accounts. So instead we switched from using Sqlite for data storage to Firebase. We chose to use Firebase because it updates and stores data in the cloud and Android Studio also supports it as it is easy to integrate it into our development progress.

At this point we have secured all relevant work and functionality to maintaining a registration and login page. Users will be able to login and register an account with their emails and create a password. The users' data will be submitted and saved within Firebase. We are now tackling get the Guest Services, Bellboy services, and Feedback pages up and running as users login in. We are also working to differentiating account types between guests and administors. The described tasks in progress (Milestones 4,5,8, and 10) are projected to be finished by the time of the first demo.

7.3 Plan of Work

Milestone 1 - Implement a Managerial Login Which Has More Security Clearances Than Regular Guest Login

Projected Due Date: 03/20/2019
Description: Manager should be able to login and view customer interaction by selecting a customer in database and pull tables of info from it. Primarily for hotel security and app maintenance as well to make sure no requests go unnoticed/unfulfilled.

Milestone 2 - Design A Page For Remote Check-In/Out

Projected Due Date: 03/24/2019
Description: In order to implement remote functionality for guest's convenience, we need to first create a display template for the users to choose between the different options. These options are included below.

Milestone 3 - Implement Remote Functionality For Guest's Convenience

Projected Due Date: 03/24/2019
Description: At the guest's convenience, they can check in/out and request a digital key if they would like to help make their stay more comfortable. Check in/check out application will send a request to the appropriate staff who will be ready for a guest checking in/ dropping off key. Digital key will generate a key in the backend, store it in the database and send a copy to guest.

Milestone 4 - Design Pages For Guest Services

Projected Due Date: 03/31/2019
Description: After guest has checked in, they should be able to utilize services that the hotel provides. Displays without functionality will be needed before we link to the app and start backend. Included as below.

- Room Service
- Maintenance Request
- Bellboy Service
- Driver/Taxi request

Milestone 5 - Implement Guest Services Functionality

Projected Due Date: 03/31/2019
Description: Since each service available is unique and has its own responsibilities, we have to implement a backend system which will take the data received in the app, cross check with database tables , and output accordingly.

Milestone 6 - Design Pages For Static FAQ

Projected Due Date: 04/07/2019
Description: A display page is needed to showcase answers to frequently asked questions by guests and to have available a number to reach the front desk

- Concierge Service
    - FAQ
    - Phone number to call front desk
- Hotel information
    - Restaurant menu
    - Gym hours
    - Pool/Spa hours

Milestone 7 - Implement Forms for Requests and Feedback

Projected Due Date: 04/07/2019
Description: Static pages which have the same functionality of google forms where customers can send feedback and/or ask more questions if they so choose.

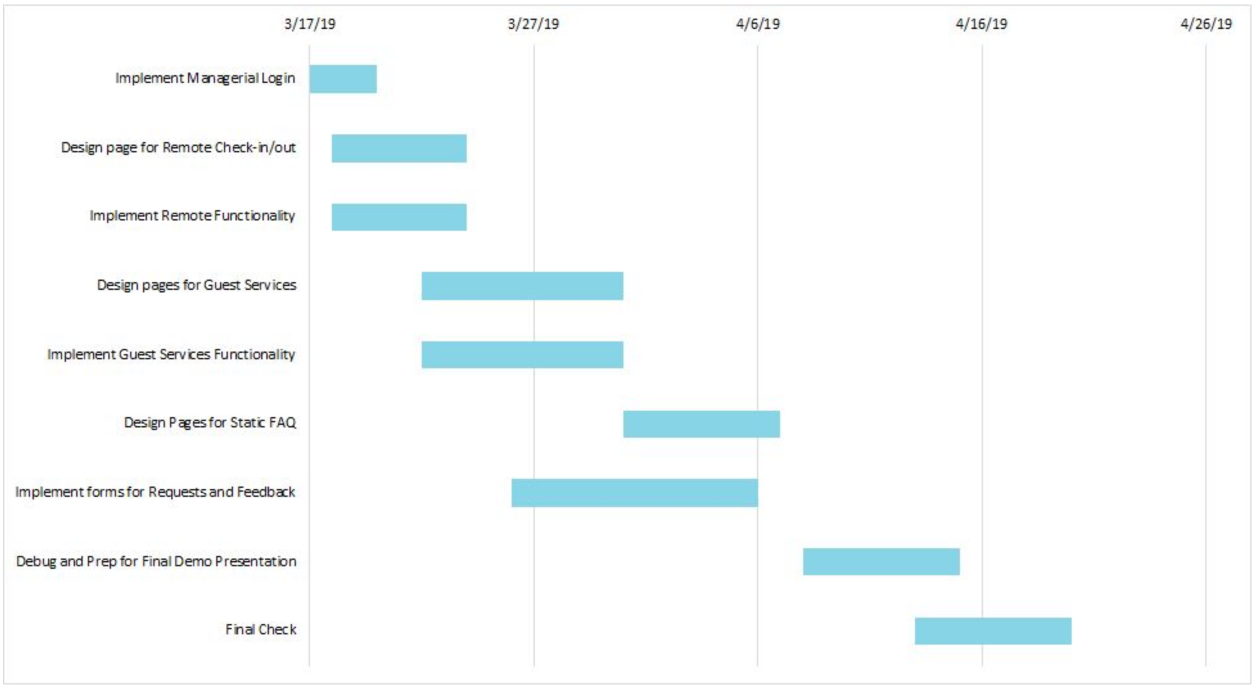
Milestone 8 - Debug and Begin Prepping Final Demo Presentation

Projected Due Date: 04/15/2019
Description: Apps will always have some mistakes which can only be identified during runtime. Our team will thoroughly test and document all possible cases of normal usage and solve any apparent errors. We will also be making a routine that the average guest would go through during their stay, this also helps with extra testing.

Milestone 9 - Final Check

Projected Due Date: 4/20/2019
Description: Our team will perform a final check on all our features and functionality of the app. We will run all possible tests to make sure that the app is 100% functional and has no problems. This milestone will be the launch of our app

7.3.1 Gantt Chart



7.4 Breakdown of Responsibilities

Sub Teams

Team A - Shilp Shah, Mandy Cheng, Eric Zhang, Keya Patel

Team B - Mohammed Sapin, Purna Haque, Thomas Tran

Team C - Mathew Varghese, Rameen Masood, Zain Sayed

Module/Class	Member/Team Responsible
Controller	Teams: A, B
Guest	Teams: C

GuestControl	Teams: C
RequestObject	Teams: A
Admin	Teams: B
RoomObject	Teams: A, B
ReservationControl	Teams: A, C
DatabaseHelper	Teams: A, B, C

**Subclasses which fall under/involve these modules are assigned to individual groups so the the overall module/class will be headed by all groups involved as it is imperative that overlapping classes must be able to work in conjunction with each other

7.4.1 Integration

In order to integrate all of these classes and their subclasses, we are using GitHub. This ensures that no code is overwritten/deleted and that all working files, such as Java code and XML files, are available for everyone to work with. Allowing people to create and merge branches together keeps everyone updated with the most current version of our project, which is important with such a large group. Utilizing GitHub desktop also streamlines our capability of viewing code as well as if any bugs get in, we can see who commits to help determine where the problem originates from.

7.4.2 Integration Testing

Unit Tests for the above classes will be written by the team developing it since they have the most knowledge about how their code works. By use of these integration tests, we make sure that these individuals models are fully functional even before committing them to the master branch. If a branch has been committed, we can assume that it passes all integration and unit tests. And so, after these tests have been performed and passed, the master branch should be able to seamlessly integrate with the new module/class. Ideally with no bugs in the code. There is also no integration co-ordinator for these tests so teams should handle the testing by themselves.

Chapter 8

References

SQLite - Create a Relationship
https://www.quackit.com/sqlite/tutorial/create_a_relationship.cfm
Advantages of Service Oriented Architecture
https://techspirited.com/advantages-disadvantages-of-service-oriented-architecture-soa
10 Common software architectural patterns
https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013

Android Architecture Patterns
https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6

Unit Testing
http://softwaretestingfundamentals.com/unit-testing/
Test Coverage
https://www.guru99.com/test-coverage-in-software-testing.html
Integration Testing
http://softwaretestingfundamentals.com/integration-testing/